Appendix B

Verifying TRUMP with Spin

The TRUMP transport protocol was designed primarily as a rate-based protocol which would fit into the rate-based congestion control framework described in this thesis. Other functionality apart from a rate-based flow control are secondary to the thesis. However, in the author's opinion, it is still essential that TRUMP be proved to be a correct and logically consistent protocol.

This appendix describes the validation of the TRUMP transport protocol using the Spin protocol verifier [Holzmann 91] [Holzmann 97] and its associated verification language, Promela. The Spin verifier is introduced, followed the Promela language and the verification capabilities of Spin. The implementation of TRUMP in Promela is then described, and the verification of this implementation using Spin is finally considered.

B.1 Introduction to Spin

The abstract from the 'Basic Spin Manual', part of the Spin software distribution, gives this description of Spin:

> Spin is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols. The system is described in a modeling language called Promela. The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered).
>
> Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs an efficient online verification of the system's correctness properties. During simulation and verification Spin checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to verify the correctness of system invariants, it can find non-progress execution cycles, and it can verify correctness properties expressed in next-time free linear temporal logic formulae.
>
> The verifier is setup to be efficient and to use a minimal amount of memory. An exhaustive verification performed by Spin can establish with mathematical certainty whether or not a given behavior is error-free. Very large verification problems, that can ordinarily not be solved within the constraints of a given computer system, can be attacked with a frugal "bit state storage" technique,

also known as *supertrace*. With this method the state space can be collapsed to a small number of bits per reachable system state, with minimal side-effects.

B.2 The Promela Language

In order to be able to read and understand the implementation of TRUMP in the Promela language, given later in this appendix, an brief introduction to the language must first be given. The following is abstracted from the 'Basic Spin Manual', with additional text relating the discussion to the TRUMP code.

Promela is a verification modeling language. It provides a vehicle for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction. The intended use of Spin is to verify fractions of process behavior, that for one reason or another are considered suspect. The relevant behavior is modeled in Promela and verified. A complete verification is therefore typically performed in a series of steps, with the construction of increasingly detailed Promela models. Each model can be verified with Spin under different types of assumptions about the environment (e.g., message loss, message duplications etc). Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification of all subsequent models.

Promela programs consist of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

B.2.1 Executability

In Promela, there is no difference between conditions and statements: even isolated boolean conditions can be used as statements. The execution of every statement is conditional on its executability. Statements are either executable or blocked. The executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. A condition can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does.

B.2.2 Data Types

The primitive data types in Promela are `bool`, `byte`, `short`, `int` and `mtype`. A `bool` is a single bit of information. A `byte` is an unsigned quantity that can store a value between 0 and 255. `shorts` and `ints` are signed quantities that differ only in the range of values they can hold (typically 16-bits and 32-bits in size, respectively).

An `mtype` variable can be assigned symbolic values that are declared in an `mtype={...}` statement. This is the preferred way of specifying message types since it abstracts from the specific values to be used, and it makes the names of the constants available to an implementation, which can improve error reporting.

Variables can be also declared as arrays, using a C style syntax. An array of size N has elements numbered 0 to N-1. Element references outside of this range cause run-time errors in Spin. C style structures can be created with the `typedef` operator: the TRUMP implementation uses a structure for the messages passed between the Sender and Receiver process. Fields within structures are accessed using the usual C style syntax.

B.2.3 Processes

The state of a variable or of a message channel can only be changed or inspected by a *process*. The behavior of a process is defined in a `proctype` declaration. The declaration body (enclosed in curly braces) consists of a list of zero or more declarations of local variables and/or statements. Promela accepts two different statement separators: an arrow '`->`'and the semicolon ';'. The two statement separators are equivalent. The arrow is used as an informal way to indicate a causal relation between two statements.

A proctype definition only declares process behavior, it does not execute it. When simulated with Spin, the `init` process is always instantiated. Statements in `init` can be used to instantiate other processes. In the TRUMP implementation, the Sender, Receiver and two link processes are instantiated via `init`.

B.2.4 Atomic Sequences

Unless blocked on a condition which is not yet true, a process can execute its statements. During verification, Spin will simulate all possible interleaving of the statements from the running processes. By prefixing a sequence of statements enclosed in curly braces with the keyword `d_step`, the user can indicate that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. It causes a run-time error if any statement in the sequence can block. `d_step` sequences in the TRUMP implementation are used to atomicise the complex manipulation of local variables.

B.2.5 Message Channels

Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, for instance as follows:

```
chan hold = [4] of { trumphdr };
```

This declares a channel that can store up to 4 messages of type `trumphdr` (a structure). Channel names can be passed from one process to another via parameters in process instantiations.

The statement `hold!expr` sends the value of variable `expr` to the channel just created: that is, it appends the value to the tail of the channel. `hold?msg` receives the message: it retrieves it from the head of the channel, and stores it in a variable `msg`. The channels pass messages in first-in-first-out order.

The send operation is executable only when the channel addressed is not full. The receive operation, similarly, is only executable when the channel is non empty. Nothing bad will happen if a statement happens to be non-executable. The process trying to execute it will be delayed until the statement, or, more likely, an alternative statement, becomes executable.

Several additional functions on channels are predefined, which allow a process to inspect the size of a channel's queue without blocking: `len(q)`, `empty(q)`, `nempty(q)`, `full(q)` and `nfull(q)`, where `q` is the name of the channel. The latter four functions return true if the channel is empty, not empty, full, or not full, respectively.

B.2.6 Control Flow

Between the lines, we have already introduced three ways of defining control flow: concatenation of statements within a process, parallel execution of processes, and atomic sequences. There are three other control flow constructs in Promela to be discussed. They are case selection, repetition, and unconditional jumps.

The simplest construct is the selection structure. Using the relative values of two variables a and b to choose between two options, for instance, we can write:

```
if
:: (a != b) -> option1;
:: (a == b) -> option2;
fi;
```

The selection structure contains two execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a *guard*.

In the above example the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected nondeterministically. If all guards are unexecutable the process will block until at least one of them can be selected. There is no restriction on the type of statements that can be used as a guard.

A logical extension of the selection structure is the repetition structure. As above, only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated.

```
do
:: (a != b) -> option1;
:: (a == b) -> option2;
od;
```

The normal way to terminate the repetition structure is with a `break` statement placed in one of the options. Note that to ensure the repetition is terminated, the guards on all the other options must be false, and hence not executable. Another way to break the repetition is with an unconditional jump: the infamous `goto` statement. The `goto label` statement is always executable, and takes execution to the named label elsewhere in the process.

One Promela statement commonly used with selection and repetition is the `skip` statement which is always executable and performs no action. A typical selection statement with no 'else' clause can be written as

```
if
:: (a == b) -> option2;
:: (a != b) -> skip;
fi;
```

Without the `skip` option, the selection would block the process until the guard `(a == b)` was indeed true. Promela also provides an `else` guard, which is only true if all other guard statements are false.

B.2.7 Promela Verification Support

An important language construct in Promela that needs little explanation is the `assert` statement. Statements of the form `assert(any_boolean_condition)` are always executable. If the boolean condition specified holds, the statement has no effect. If, however, the condition does not necessarily hold, the statement will produce an error report during verifications with Spin.

When Promela is used as a verification language the user must be able to make very specific assertions about the behavior that is being modeled. In particular, if a Promela is checked for the presence of deadlocks, the verifier must be able to distinguish a normal end state from an abnormal one.

A normal end state could be a state in which every Promela process that was instantiated has properly reached the end of the defining program body, and all message channels are empty. But, not all Promela process are, of course, meant to reach the end of their program body. Some may very well linger in an idle state, or they may sit patiently in a loop ready to spring into action when new input arrives.

To make it clear to the verifier that these alternate end states are legal, and do not constitute a deadlock, a Promela model can use end state *labels*. There may be more than one end state label per verification model. If so, all labels that occur within the same process body must be unique. The rule is that every label name that starts with the three character sequence "end" is an endstate label.

In the same spirit as the end state labels, the user can also define progress state labels. In this case, a progress state label will mark a state that must be executed for the protocol to make progress. Any infinite cycle in the protocol execution that does not pass through at least one of these progress states is a potential starvation loop. Again, all progress labels that occur within the same process body must be unique. The rule is that every label name that starts with the character sequence "progress" is an progress state label.

B.3 Spin Verification Capabilities

The verification capabilities of Spin are described in detail in [Holzmann 91] and in the 'Basic Spin Manual'. Again, the following brief description is abstracted from the latter.

Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs a fast exhaustive verification of the system state space. The verifier can check, for instance, if user specified system invariants may be violated during a protocol's execution.

If invoked as

```
$ spin -a protocol.spin
```

Spin generates a protocol analyzer for the protocol described in the Promela language in the `protocol.spin` file. The output analyser is written into a set of C files that can be compiled to produce the analyzer (which is then executed to perform the analysis). To guarantee an exhaustive exploration of the state space, the program is compiled simply as

```
$ cc -o run pan.c
```

The executable program run can now be executed to perform the verification. The verification is truly exhaustive: it tests all possible event sequences in all possible orders. Exhaustive search works up to system state spaces of roughly 100,000 states. For larger systems this may, however, exhaust the available memory on the machine used.

The coverage of a conventional analysis goes down rapidly when the memory limit is hit, i.e. if there are twice as many states in the full state space than we can store, the effective coverage of the search is only 50% and so on. Spin does substantially better in those cases by using the bit state space storage method [Holzmann 90]. Large to very large systems can still be analyzed by using a memory efficient bit state space method by compiling the analyser as

```
$ cc -o run -DBITSTATE pan.c
```

An indication of the coverage of such a search can be derived from the hash factor given in the analyser's output. The hash factor is roughly equal to the maximum number of states divided by the actual number of states. A large hash factor (larger than 100) means, with high reliability, a coverage of 99% or 100%. As the hash factor approaches 1, the coverage approaches 0%.

Note carefully that the analyzer realizes a partial coverage only in cases where traditional verifiers are either unable to perform a search, or realize a far smaller coverage. In no case will Spin produce an answer that is less reliable than that produced by other automated verification systems (quite on the contrary).

B.4 Implementing TRUMP in Promela

TRUMP is implemented in Promela as several intercommunicating processes. The Sender process attempts to deliver data packets successfully to the Receiver, following the protocol rules for connection setup, data transmission and connection teardown. The Receiver process receives data packets from the Sender, and returns acknowledgment packets. The Sender and Receiver are connected via two unidirectional links, whose operations are performed by two instantiations of the Badlinksrc process.

As the goal here is to verify the corrctness of the TRUMP protocol, features of TRUMP which do not affect its protocol operation are not implemented. This includes the protocol's rate-based flow control, the security features of TRUMP, and the use of connection identifiers. These features have no impact on the logical correctness and consistency of the protocol.

Protocol features of TRUMP which have not been implemented are the qualities of service and packet fragmentation/reassembly. The reason for these omissions is the limit on the time available to complete this thesis. However, despite this, the majority of the TRUMP protocol has been implemented in Promela and verified by the Spin program.

Although TRUMP is not as large as a 'heavyweight' protocol such as TCP, it is still a complex protocol, and cannot be exhaustively verified using the platforms available to the author, mainly due to memory size considerations. To prove that TRUMP is indeed correct where only some of the features are available, the implementation of TRUMP in Promela is controlled by macro processor[1] 'define's which enable or disable certain features:

REORD: Allow packet reordering on the two links betwen the Sender and the Receiver. The maximum number of reordering operations per link is given by the value MAXREORD in a `constants` file. The maximum number of packets 'buffered' by the link for reordering is given by the value MAXPKTS in the `constants` file.

LOSE: Allow packet loss on the two links betwen the Sender and the Receiver. The maximum number of packet losses per link is given by the value MAXLOSSES in the `constants` file.

SELACK: Allow selective acknowledgments to be used between the Sender and the Receiver. The number of packets acknowledged in each selective acknowledgment is given by the value SELACKSIZE in the `constants` file.

RECVERRS: Allow the Receiver to send errors in Special Acknowledgment at any time.

IMPLICIT: Allow implicit connection setups and teardowns as well as the default explicit connection setups and teardowns.

ASSERT: Install many Promela `assert()` statements into the TRUMP implementation.

The **ASSERT** option adds assertions to TRUMP and does not increase its complexity. The **RECVERRS** and **IMPLICIT** options add many extra states to both the Sender and Receiver processes, and increase the size of the state space to be exhaustively searched and verified. The same is true for the **SELACK** option, but this also increases the size of the inter-process messages and the number of local variables, which again increases the state space. The **REORD** and **LOSE** options do not affect the Sender or Receiver, but the packet reorderings and losses greatly add to the size of the state space to be searched.

B.4.1 TRUMP's Promela Implementation

TRUMP's implementation in Promela is divided into two files: the actual implementation is contained in `trump.spin`, and the constants used by TRUMP are contained in the `constants` file. This file is short and is given below.

```
/******************************/
/** Easily-changed constants **/
/******************************/
#define DATAPKTS        1       /* Send this many data packets */

#ifdef REORD
# define MAXPKTS        4       /* Max packets buffered by reordering */
# define MAXREORD       20      /* Max number of reorders permitted */
#endif

#ifdef LOSE
# define MAXLOSSES      20      /* Maximum number of losses permitted */
#endif
```

[1]The macro processor used is the C preprocessor.

```
#ifdef SELACK
# define SELACKSIZE     8       /* Number of acks in selack packet */
# define EMPTY          255     /* Ack pkt is empty if seq_base==EMPTY */
#endif

/* The Sender keeps a linked list of
 * the in-transit packets. Each element
 *  has one of the following values.
 */
#define NOTSENT         253     /* Packet has not been transmitted yet */
#define RECEIVED        254     /* Packet was sucessfully received */
#define NULL            255     /* Null pointer */
```

The implementation of TRUMP in `trump.spin` is so heavily affected by macro processor directives as to be virtually unreadable. The version of the file below gives the values of the macro processor options, the code after macro processing, and with line numbers.

Some notes should be first given on the TRUMP implementation. In order to determine which packet to transmit next (and hence its sequence number), the TRUMP Sender needs to keep a linked list of in-transit packets and not-yet-transmitted packets. Promela does not have the capability of representing arbitrarily-sized linked lists. To overcome this deficiency, the Sender declares an array of bytes, `data_status`, to store the list of in-transit packets.

Two byte variables, `head` and `tail`, point at the head and tail nodes of the list. An arbitrary value of 255 is chosen to represent the NULL pointer. Elements in the array hold the index number of the next element in the array. The tail node has the value NULL.

There will obviously be some array elements which are not in transit to the Receiver. These have either the value NOTSENT (253) if they have not been transmitted yet, or the value RECEIVED (254) if they were sucessfully received by the Receiver.

As values in the `data_status` array above 252 have special meanings, there cannot be elements in the array with index 252 or above. Therefore, the TRUMP implementation in Promela is limited to around 250 data packets.

Here, finally, is the implementation of TRUMP in `trump.spin`.

```
/**
 ** TRUMP verification in Promela
 **
 ** Copyright (c) 1997 Warren Toomey
 ** Revision: 1.32
 ** Date: 1997/04/17
 **/


/*************************/
/** Source-code defines **/
/*************************/

#define REORD                            /* Allow packet reordering on links */
#define LOSE                             /* Allow packet loss on links */
#define SELACK                           /* Allow selective acknowledgments */
#define RECVERRS                         /* Allow receiver to send errors */
```

```
#define IMPLICIT                        /* Allow implicit setups/teardowns */
#define ASSERT                          /* Install lots of assertions */

#include "constants"                    /* Get constants used below */


/*******************/
/** TRUMP Headers **/
/*******************/

/* Each packet must be one of these types */
mtype = { setup, sack, data, ack, teardown, datasyn }


/* A TRUMP packet has the following headers */
typedef trumphdr {
    mtype pkt_type;                     /* Type of packet this is */
    byte  seq_no;                       /* Sequence number of data segment */
                                        /* in selack[0] */
    byte  seq_top;                      /* Top seq# in selack bitmap */
    bool  selack[SELACKSIZE];           /* Selack bitmap */
    bool  explicit_down;                /* If true, do explicit teardowns */
    bool  error_val;                    /* Error value of reply */
}

/***********************/
/* Channel Definitions */
/***********************/

chan sendtolink = [1] of { trumphdr }   /* Channel from src to badlink  */
chan recvtolink = [1] of { trumphdr }   /* Channel from dest to badlink */
chan linktorecv = [1] of { trumphdr }   /* Channel from badlink to dest */
chan linktosend = [1] of { trumphdr }   /* Channel from badlink to src  */


/*********************/
/** Network Section **/
/*********************/

proctype Badnetwork(chan in, out)
{
  trumphdr packet;                      /* Packet received on link */

  chan hold = [MAXPKTS] of { trumphdr }; /* Pkt buffer for reordering */
  xr hold; xs hold;
  byte reordcnt;                        /* Count of # of reorderings done */


  byte losecnt;                         /* Count of # of losses done */
  losecnt=0;                            /* Initialise loss count */
  reordcnt=0;                           /* Initialise reordering count */

end_bad0: /* Forerver, do */
        do
        :: in?packet ->         /* Receive a packet from in channel */
                if
                :: out!packet   /* Retransmit it immediately, */

```

```
                :: ((reordcnt<MAXREORD) && nfull(hold)) ->
                    hold!packet; reordcnt++; /* save it for later transmit, */

                :: (losecnt<MAXLOSSES) -> losecnt++; /* or lose it entirely */
                fi;

        :: hold?packet ->
                out!packet      /* Reinsert old data if no link pkt */
        od;
}


/******************/
/** TRUMP Sender **/
/******************/

/* The sender has the following main states:
 *
 *  transmitting setup packets (optional)
 *  transmitting data packets
 *  transmitting teardown packets
 *  finished!
 */

proctype Sender(chan send_in, send_out)
{
 trumphdr a;                    /* Packet received from destination */
 byte data_status[DATAPKTS];    /* List of in-transit seq nums */
 byte head;                     /* Head of list */
 byte tail;                     /* Tail of list */
 byte prev;                     /* Previous node pointer */
 byte lost;                     /* Highest known lost packet */
 byte notsent;                  /* Lowest not-yet sent packet */
 byte i;                        /* Loop counter */
 byte S;                        /* Next sequence number to send */
 bool explicit_conn;            /* Should we do an explicit connection? */
 bool explicit_down;            /* Should we do an explicit teardown? */


 /* Initialise the data_status array to all NOTSENT */
 d_step {
   i=0; S=0; head=NULL; tail=NULL; lost=NULL; notsent=0;
   do
   :: (i==DATAPKTS) -> break;
   :: (i!=DATAPKTS) -> data_status[i]= NOTSENT; i++;
   od;
 }

 /* Choose to do an implicit or explicit connection */
 if
 :: (1) -> explicit_conn=0;                     /* Implicit connection */
           if
           :: (1) -> explicit_down=0;           /* And implicit teardown, or */
           :: (1) -> explicit_down=1;           /* Explicit teardown */
           fi;

           /* Send off the implicit conn packet, which is also 1st data */
           d_step {
```

```
            a.pkt_type=datasyn; a.error_val=0; a.seq_no=0;
            a.explicit_down=explicit_down; head=0; tail=0; lost=NULL;
            data_status[0]=NULL; notsent=1;
           };

           send_out!a; goto progress_send0;     /* Move to state 2, data tx */
 :: (1) -> explicit_conn=1; explicit_down=1;    /* Explicit connection */
 fi;


 /* State 1: sending setup packets */
 do
 :: send_in?a -> /* Receive a packet from the receiver */
         if /* it's a sack with no error, move to state 2 */
         :: (a.pkt_type==sack && a.error_val==0) -> break;

            /* it's a sack with an error, terminate now */
         :: (a.pkt_type==sack && a.error_val!=0) -> goto end_exit_state;

            /* some other packet, ignore it! */
         :: else ->
                assert(0);      /* Should not get any other type of packet */
         fi;

        /* If we can't receive a packet from the receiver */
        /* then send it a setup packet */
 :: (empty(send_in) && nfull(send_out)) ->
          d_step { a.pkt_type=setup; a.seq_no=0; a.error_val=0; }
          send_out!a;
 od;

progress_send0:


 /* State 2: sending data packets */
 do
 :: send_in?a; /* Receive a packet from the receiver */
         if /* it's a sack with no error, ignore it */
         :: (a.pkt_type==sack && a.error_val==0) -> skip;

            /* it's a sack with an error, terminate now */
         :: (a.pkt_type==sack && a.error_val!=0) -> goto end_exit_state;

            /* it's an ack packet, update the data_status array */
         :: (a.pkt_type == ack) ->
                d_step {
                  printf("Sender got SELACK %d/%d \n",a.seq_no,a.seq_top);
                  i=a.seq_no; S=0; printf("\t");
                  do
                  :: (i>a.seq_top) -> break;
                  :: else -> printf("%d ", a.selack[S]); i++; S++;
                  od; printf("\n");
                }

                /* Assert that  seq_no <= seq_top < seq_no+SELACKSIZE */
                /* We could assert that no ack bits above seq_top are on, */
                /* but the sender ignores them anyway */
                assert(a.seq_no<=a.seq_top);
```

```
assert(a.seq_top<(a.seq_no+SELACKSIZE));

/* If it's an ack for the datasyn, check the explicit_down */
d_step {
  if
  :: ((explicit_conn==0)&&(a.seq_no==0)) ->
    if
    :: ((explicit_down==0)&&(a.explicit_down==0)) -> skip;
    :: ((explicit_down==0)&&(a.explicit_down==1)) ->
        explicit_down=1; /* Recvr asked for explicit teardown */

    :: ((explicit_down==1)&&(a.explicit_down==0)) ->

        /* Receiver asked for implicit teardown, but we've */
        /* already chosen explicit teardown, error! */
        assert(0);
    :: ((explicit_down==1)&&(a.explicit_down==1)) -> skip;
    fi;
  :: else -> skip;
  fi;
}

/* Deal with each sequence number in turn */
d_step {
   S=0;         /* S points at 0th bit in bitmap */
   /* For the selack code, loop a.seq_no up to a.seq_top */
   do
   :: (a.seq_no > a.seq_top) -> break;
   :: else ->

        /* Assert that we did in fact transmit this one */
        assert(data_status[a.seq_no]!=NOTSENT);

        if
        :: (a.selack[S]==0) ->  skip; /* Ignore lost pkts */
        :: (a.selack[S]!=0) ->  /* it was received! */

        /* Assert that we did in fact transmit this one */
                assert(data_status[a.seq_no]!=NOTSENT);

                if
                :: (head==NULL) -> skip; /* Already got it */
                :: (data_status[a.seq_no]==RECEIVED) -> skip;

                :: (head==a.seq_no) -> /* it's head packet */
                    /* Move the head up */
                    head= data_status[a.seq_no];

                    if /* Null lost ptr if we received it */
                    :: (lost==a.seq_no) -> lost=NULL;
                    :: else -> skip;
                    fi;

                    if /* list now empty? */
                    :: (head==NULL) -> tail=NULL;
                    :: else -> skip;
                    fi;
                :: else ->
```

```
                                /* Not the head! Find previous node */
                                prev=head;

                                do  /* prev can't point at NULL! */
                                :: (data_status[prev]==NULL) -> assert(0);
                                :: (data_status[prev]==a.seq_no) -> break;
                                :: else -> prev= data_status[prev];
                                od;

                                if /* Null lost ptr if we received it */
                                :: (lost==a.seq_no) -> lost=prev;
                                :: else -> skip;
                                fi;

                                /* Point lost at prev if prev's */
                                /* seq_no < a.seq_no */
                                if
                                :: (prev<a.seq_no) -> lost=prev;
                                :: else -> skip;
                                fi;
                                /* Now, remove seq_no from list */
                                data_status[prev]=data_status[a.seq_no];
                            fi;
                            if /* seq_no was the tail, point tail at prev */
                            :: (a.seq_no==tail) -> tail=prev;
                            :: else -> skip;
                            fi;

                            /* Mark the sequence number as RECEIVED */
                            data_status[a.seq_no]=RECEIVED;
                    fi;
                    a.seq_no++; S++;        /* Move up to the next seq_no */
               od;
            };
        /* some other packet, error! */
     :: else -> assert(0);
     fi;

    /* If we can't receive a packet from the receiver */
    /* then send it a data packet */
:: (empty(send_in) && nfull(send_out)) ->

      /* Complicated bit. To obtain the next sequence number: */
      /*   If lost!=NULL, choose head of intransit list, else */
      /*   Use lowest notsent packet */
      d_step {
        S= NULL;            /* Assume no packets left to send */

        if /* Lost can't point to anything is list is empty! */
        :: ((head==NULL) && (lost!=NULL)) -> assert(0);
        :: else -> skip;
        fi;

        if  /* no packets left to send */
        :: ((lost==NULL)&&(head==NULL)&&(notsent==DATAPKTS)) -> skip;

        :: ((lost==NULL)&&(notsent<DATAPKTS)) -> /* choose new seq# */
            S= notsent;             /* Notsent has lowest notsent seq# */
```

```

        if                      /* Append notsent to tail */
        :: (tail==NULL) -> head=notsent;
        :: (tail!=NULL) -> data_status[tail]=notsent;
        fi;
        tail=notsent; data_status[tail]=NULL; notsent++; /* Link ops */

      :: else ->          /* Something must be left in intransit list */

        S= head;

        if              /* Remove head node */
        :: (head==tail) -> head=NULL; tail=NULL;

        :: (head!=tail) ->
            assert(head!=NULL); assert(tail!=NULL);
            head= data_status[head];
        fi;

        if                      /* Append S to tail */
        :: (tail==NULL) -> head=S; tail=S;

        :: (tail!=NULL) -> data_status[tail]=S; tail=S;
        fi;

        data_status[tail]=NULL;

        if /* we're retxing the highest lost packet */
        :: (lost==S) -> lost=NULL;      /* No lost pkts left now */
        :: else -> skip;
        fi;
      fi;
    } /* End of the d_step many lines ago! */

    /* If S is still NULL here, no pkts to tx, then move to next state */
    if
    :: (S==NULL) -> break;
    :: else -> skip;
    fi;

    d_step { /* otherwise build and transmit the data packet */
      a.pkt_type=data; a.error_val=0; a.seq_no=S;
    }
    send_out!a;
 od;

progress_send1:
  /* Assert that all packets marked as RECEIVED */
  i=0;
  do
  :: (i==DATAPKTS) -> break;
  :: (i!=DATAPKTS) -> assert(data_status[i]==RECEIVED); i++;
  od;

  if /* we don't need to do an explicit teardown, skip the teardown state */
  :: (explicit_down==0) -> goto end_exit_state2;
  :: else -> skip;      /* we do, so go straight into it */
  fi;
```

```


 /* State N: sending teardown packets */
 do
 :: send_in?a -> /* we can receive a packet from the receiver */
         if /* it's a teardown, move to exit state */
         :: (a.pkt_type==teardown) -> break;

            /* some other packet, ignore it! */
         :: (a.pkt_type != teardown) -> skip;
         fi;

      /* If we can't receive a packet from the receiver */
      /* then send it a teardown packet */
 :: (empty(send_in) && nfull(send_out)) ->
         d_step { a.pkt_type=teardown; a.seq_no=0; a.error_val=0; }

         send_out!a;
 od;

end_exit_state:
 printf("Sender exiting\n");

/* Sit here mopping up any packets */
end_exit_state2:
 do
 :: send_in?a;
 od;
}


/********************/
/** TRUMP Receiver **/
/********************/

proctype Receiver(chan recv_in, recv_out)
{
 trumphdr b;                    /* Received packet */
 byte i;                        /* Loop counter */
 trumphdr a;                    /* Acknowledgment packet */
 bool alreadygot;               /* True if already marked in selack packet */
 bool explicit_down;            /* Expect an explicit teardown */
 bool gotit[DATAPKTS];          /* True if we've got this seq# */
 byte got_thismany;             /* Count of seq#s we've got (excl. dups) */

                                /* Initialise the empty acknowledgment packet */
 d_step {
    a.seq_no= EMPTY;
    a.pkt_type=ack;

    i=0;                        /* Initialise the gotit array */
    got_thismany=0;
    do
    :: (i==DATAPKTS) -> break;
    :: (i!=DATAPKTS) -> gotit[i]=0; i++;
    od;
 };

```

```
                /* State 1: receiving setup packets */
 do
 :: recv_in?b;
    if
    :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return sack pkt */
        d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=0; explicit_down=1; }

        recv_out!b; break;

    :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return ERROR sack pkt */
        d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=1; }

        recv_out!b;
        goto end_recv_tosspkts; /* and just sink remaining packets */

    :: (b.pkt_type==datasyn) -> /* Implicit conn pkt */
        explicit_down=b.explicit_down; /* Keep explicit teardown flag */

        /* Choose for explicit or implicit teardown */
        if
        :: (b.explicit_down==1) -> skip;
        :: (b.explicit_down==0) -> explicit_down=0;     /* Implicit */
        :: (b.explicit_down==0) -> explicit_down=1;     /* Explicit */
        fi;
        goto recv_gotdata; /* Skip to the data receive state */

    :: else -> skip;    /* If not a setup packet, ignore it */
    fi;
  od;

progress_recv0:
                /* State 2: receiving data packets */
 do

 :: (got_thismany==DATAPKTS) ->

end_recvimplicit:
    recv_in?b; goto process_data;       /* End state if got all seq#s */
 :: (got_thismany!=DATAPKTS) ->         /* otherwise it isn't */

    recv_in?b;

process_data:
    if
    :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return sack pkt */
        d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=0; }
        recv_out!b;

    :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return ERROR pkt */
        d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=1; }
        recv_out!b;
        goto end_recv_tosspkts; /* go and sink remaining packets */

    :: (b.pkt_type==data) -> /* Received a data packet */
recv_gotdata:
                                /* Mark seq# as seen */
        if
        :: (gotit[b.seq_no]==1) -> skip;
```

```
                             /* & count the # unique seq#s we've received */
        :: (gotit[b.seq_no]==0) -> gotit[b.seq_no]=1; got_thismany++;
        fi;

        /* First thing is to see if we can fit this seq_no */
        /* into the existing acknowledgment packet */
        if
        :: ((a.seq_no!=EMPTY) && ((b.seq_no<a.seq_no)
            || (b.seq_no>=a.seq_no+SELACKSIZE))) ->
                /* Seq_no doesn't fit, so transmit existing ack packet */
                recv_out!a;

                /* Reinitialise the ack packet */
                d_step { a.seq_no= EMPTY; }
        :: else -> skip;
        fi;

        /* At this point, either the ack packet is EMPTY or we can */
        /* fit the seq_no in */
        d_step {
          if
          :: (a.seq_no==EMPTY) ->
                a.seq_no= b.seq_no; a.seq_top= b.seq_no;
                a.selack[0]=1; alreadygot=0;
                a.explicit_down=explicit_down;

          :: (a.seq_no!=EMPTY) ->
                i = b.seq_no - a.seq_no;
                alreadygot=a.selack[i]; a.selack[i]=1;
                if
                :: (b.seq_no>a.seq_top) -> a.seq_top=b.seq_no;
                :: else -> skip;
                fi;
          fi;
        }

        /* If data pkt duplicates one we already have, send ack pkt back NOW */
        if
        :: (alreadygot==1) ->
                recv_out!a; a.seq_no=EMPTY;
        :: else -> skip;
        fi;

    :: (b.pkt_type==teardown) -> /* Received a teardown packet */

        /* It's an error if implicit teardowns selected */
        assert(explicit_down==1);

        /* Reflect the teardown packet */
        d_step { b.pkt_type=teardown; b.seq_no=0; b.error_val=0; }
        recv_out!b; break;

    :: (b.pkt_type==data) -> /* If pkt is a data pkt, return ERROR sack pkt */
        d_step { b.pkt_type=sack; b.error_val=1; } /* (to be perverse) */

        recv_out!b; goto end_recv_tosspkts;

    :: else -> skip;    /* Not a setup, data or teardown packet, ignore it */
```

```
     fi;
   od;


                /* State 3: Got a teardown, mopping up */
end_recv0:
 do
 :: recv_in?b;
     if
     :: (b.pkt_type==teardown) -> /* Reflect teardowns to source */
         d_step { b.pkt_type=teardown; b.seq_no=0; b.error_val=0; }
         recv_out!b;

     :: else -> skip;    /* If not a setup or teardown packet, ignore it */
     fi;
 od;

                /* State for collecting remaining packets */
end_recv_tosspkts:
  do
  :: recv_in?b; d_step { b.pkt_type=sack; b.error_val=1; } recv_out!b;
  od;
}


/***************************/
/** System Initialisation **/
/***************************/

init
{
  printf("TRUMP verification in Promela\n");
  run Sender(linktosend, sendtolink);
  run Badnetwork(sendtolink, linktorecv);
  run Badnetwork(recvtolink, linktosend);
  run Receiver(linktorecv, recvtolink);
}
```

B.5 TRUMP State Diagrams

Much of the Promela code for TRUMP deals with the manipulation of internal variables to detect packet loss and duplication, and to ensure packet retransmission when required. This tends to obscure the operation of TRUMP as a protocol, i.e as a communications language between two peers. To expose the operation of the protocol itself, state diagrams for the TRUMP sender and receiver are presented: these emphasise the communication operations in TRUMP. States and transitions dealing with the 'internal' functionality of TRUMP (e.g working on the linked list of in-transit packets) have been omitted to improve clarity.

The two diagrams show the states and state transitions for the TRUMP sender and receiver. States are represented as nodes with line numbers corresponding to the implementation of TRUMP given in the previous section: essentially, these correspond to the points where a choice is available, or where a implementation can be blocked on a receive or send operation. Receiving states are marked in blue, and sending states are marked in pink. Transitions are represented as directed edges, with labels describing the reason for the transition. Transitions marked with a '*' character must be taken.

for **all** possible interleavings of statements and events across all the processes in the implementation: this forms the *state space* to be verfied.

The state space must also incorporate the state of all local variables within each process, as state transitions may be chosen on the basis of these state variables. In the implementation of TRUMP in Promela, both the sender and receiver have several local variables, of which the largest are the arrays used to track the number of packets sucessfully transmitted.

In the verification of the TRUMP protocol with Spin, the number of possible states, combined with the size of each state, makes an exhaustive proof of the protocol impossible. The computing equipment available does not have enough real memory to hold the state space. Using the 'supertrace' method of verification, the amount of memory required to hold each state is much reduced, but again the number of states is too large to fully prove the protocol. However, let us look at the areas in which the protocol can be proved.

B.6.1 No Features Enabled

In Section B.4, it was mentioned that certain TRUMP features could be enabled or disabled via preprocessor macros. Each feature adds extra complexity and state size to TRUMP, and for some of the combinations of features, the TRUMP protocol can be verified.

With no features enabled, the protocol provides reliable data transmission of N data packets across an error-free link with no selective acknowledgments (i.e one acknowledgment per data packet) using explicit connection setup and teardown. For values of N up to 28, the protocol can be exhaustively proved correct: there are 177,614 states (of size 128 bytes) and 222,136 state transitions for N =28.

Above N =28, we must use the 'supertrace' method of verification in Spin. Here, the verifier can estimate the fraction of all states verified by consulting the number of unused hash entries and the number of hash collisions for a hash table that is used to represent the state space. Using the supertrace method, a protocol can be proved correct for the fraction of possible states covered by the verifier. In the case of Spin, a verification shows that a protocol is correct after covering either '$\geq$ 99.9% of states', '$\geq$ 98% of states' or '$<$ 98% of states'.

Using supertrace, TRUMP with no features enabled is provably correct after covering $\geq$ 98% of states for N up to 250 data packets.

B.6.2 Selective Acknowledgments

With the enabling of selective acknowledgments of size 8 (that is, one acknowledgment for 8 data packets), the state vector is 60 bytes larger then for the equivalent 'no features' state, and the number of states is approximately 4 times that of the equivalent 'no features' protocol. The supertrace method was used for all values of N, showing that TRUMP with selective acknowlegments of size 8 is provably correct after covering $\geq$ 98% of states for N up to 250 data packets.

B.6.3 Receiver Errors

One optional feature of the TRUMP implementation in Promela is to allow the receiver to return a special 'error' acknowledgment at any time during the course of the connection setup, data transmission and connection teardown. Upon receipt of this acknowledg-